Reconfigurable Architectures and Instruction Sets

Programmability, Code Generation, and Program Execution

Rainer Buchty

Universität Karlsruhe (TH), Institut für Technische Informatik
76128 Karlsruhe, Germany
E-Mail: `buchty@ira.uka.de`

Abstract. Within Self-reconfiguring systems two basic problems arise: on instruction level, reconfigurable instruction sets make program generation and execution inherently difficult. In addition, reconfiguration must not violate certain restrictions vital for the running application. In this paper we describe a combined low-overhead approach which targets both problems by instrumenting an attributed low-overhead run-time environment which is able to dynamically map application-specific instructions to a variety of implementation alternatives while strictly adhering to given application demands. Our approach can be used application-independent and is suitable for use within the adaptive planning stage of a Self-X system as demonstrated by a reference implementation.

1 Introduction and Motivation

Self-X capabilities are introduced into computer systems for several reasons. Existing implementations usually deal with large-scale reconfiguration dealing with availability and fault-tolerance optimization, such as task and service migration, storage systems, or network communication. These are typical problems of server and cluster computing.

If Self-X is introduced to architecture level – as required for e.g. power vs. performance optimizations in embedded systems – different problems arise. In this paper we want to concentrate on two basic problems resulting from system reconfiguration on instruction level with respect to running applications.

Firstly, reconfigurable instruction sets make program generation and execution inherently different because of the changing mapping of functionality to reconfigurable instruction opcodes. For obvious reasons it is not suitable to enforce an application recompilation with every hardware reconfiguration. Instead, we propose an easy mapping technique to map dedicated instructions to either hardware or software modules, so that this sort of translation takes place on-the-fly without requiring adopting the compiler infrastructure, application recompilation, or application stopping and resuming.

Secondly, it must be ensured that reconfiguration efforts do not harm a running application, i.e. reconfiguration must be robust with respect to application requirements such as e.g. computation accuracy or throughput. Therefore, some method must be implemented which guides the reconfiguration process to let reconfiguration only take place within given application boundaries. To solve this problem, we propose a method of attributing not only the application itself on a high-level, but also the application's building blocks, i.e. software and hardware modules.

We apply these methods to an example application where we demonstrate, how both methods can be combined into a single building block ensuring binary compatibility and being part of the system's adaptive planning infrastructure.

This paper is organized as follows: Section 2 will describe our approach in detail, whereas Section 3 illustrates an example application. The paper is concluded with Section 4.

2 Concept Outline

The concept of our approach is two-fold. Firstly, a method for maintaining binary compatibility within a system offering a reconfigurable instruction set is presented. Then, we discuss our approach on adhering to given application requirements.

2.1 Achieving Binary Compatibility

One major topic in reconfigurable architectures is the introduction of reconfigurable instruction sets to enable dynamic application-specific instruction set enhancements. First architectures were introduced in the 1990s which were either extending static instruction sets with application-specific functionality [2,1], or were data graph-oriented [6,5].

In this work we want to concentrate on the first group, i.e. static instruction sets enriched with application specific functions. With the Virtex-4FX [4,3] a commercial product exists featuring a general purpose processor core which supports dynamic instruction set extension through a dedicated interface.

The imminent problem with such extensible, reconfigurable instruction sets is how to maintain binary compatibility among various configurations of the dynamic instruction set. This problem can be targeted using Just-in-Time (JiT) translation. The Java Virtual Machine (JVM) is a well-known approach to translate an independent byte-code into the target processor's instruction set. However, Java not only defines such byte-code but also an entire run-time system, so the entire Java infrastructure becomes rather heavyweight.

Our approach condenses this concept to the core function, i.e. direct mapping of an arbitrary byte-code to the desired target representation. This process can be even done easily in hardware, especially if the input and target representation are pretty close: if only the reconfigurable part of the instruction set, i.e. the application-specific instructions (ASI), need to be mapped, the translation process will be rather simple. Hence, we divide the instruction set into a static and a reconfigurable part. On hardware level, a number of opcodes is reserved for reconfigurable instructions (providing the ASIs), triggering the execution of a desired application-specific function (ASF).

On software level, any ASF appears as a defined exception which is resolved by the translation process: if the ASF is present as a hardware implementation , the associated reconfigurable instruction (e.g. a so-called user-defined instruction, UDI, for the PPC405 core present in Xilinx Virtex-4FX FPGAs) will be executed, otherwise a call to an assigned library function or subroutine is inserted into the instruction stream. This is illustrated by Figure 1: here, the application-specific function #5 (`ASF #5`), taking two parameters and returning a single result, is executed within a small code fragment. Upon occurrence of the ASF, the following actions take place:

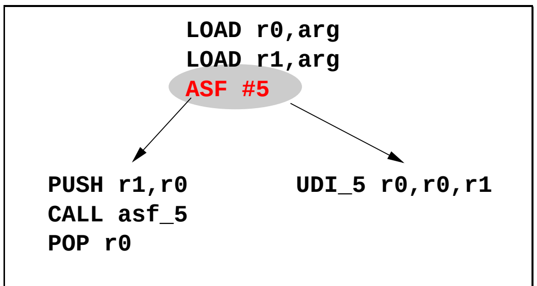

Fig. 1. Exemplary Instruction Translation Process

1. It is resolved, which functionality is associated with `ASF #5`. This is necessary, because ASFs are just enumerated during program generation. Same ASF numbers in different tasks or programs not necessarily represent same functionality.
2. It is checked, whether this function is already present in hardware, or if the associated functionality can or should be added to the instruction set as a user-defined instruction (UDI). If either is the case, `ASF #5` gets replaced with the opcode assigned to the UDI.
3. If the associated function is not present in hardware and/or can/should not be loaded then `ASF #5` gets replaced by a call to the corresponding library function. This is the case when e.g. all available UDIs are already occupied and none can be replaced without violating another task's constraints.

It must be noted, that it is possible to supply several implementations for a single ASF depending on the application demands as described in the following section. For instance, a required computation can be performed with different accuracy, i.e. single or double precision floating-point, or fixed-point/integer.

2.2 Granting Application Demands

In a reconfigurable system it must be ensured that application demands are granted, so that the application does not break because of reconfiguration. This means that any adaptive planning and hence hereon based reconfiguration must take application demands into consideration as another objective function.

For this, we like to introduce the concept of attribution. Such attribution takes place on application level as well as within the application's building blocks such as software libraries, hardware accelerators etc. This means, that the application source code is enriched by certain attributes defining the application's requirements such as required minimum throughput, data sizes, or computation accuracy. Alternative implementations of the aforementioned ASFs, in term, contain similar attributes providing information about their respective performance.

During start-up and run-time, application and ASF attributes are weighed against each other and thus appropriate ASF implementations are chosen for initial start-up and later to adopt to changed requirements. This weighing process is driven by additional objective functions and environmental influences (such as e.g. temperature or mode of power supply) and eventually leads to system reconfiguration, i.e. exchanging ASF implementations as required.

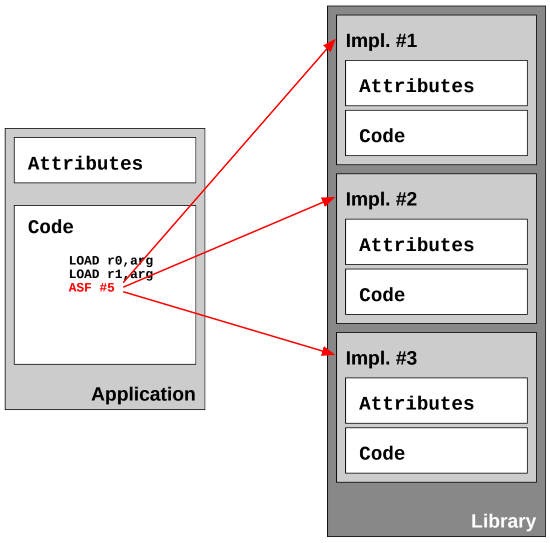

Fig. 2. Attribution Concept

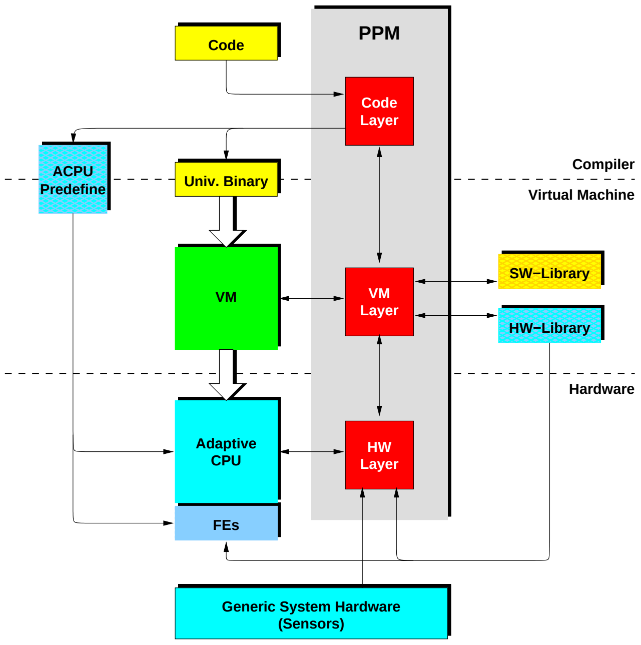

Fig. 3. Example Implementation

Figure 2 visualizes how the translation process is guided. We see an application on the left which carries code and attributes. The application contains a part invoking an ASF. Now the application attributes are weighed against the various implementations. In this little example three alternatives are given: #1 makes use of hardware acceleration by invoking a user-defined instruction. #2 and #3 provide software implementations using the fixed instruction set and perform the computation with different precision. Depending on the outcome of the weighing process, one of these three implementations will be selected and inserted into the instruction stream.

3 Example Implementation

We applied the aforementioned concepts to the design of an architecture focusing on power vs. performance optimization. This example implementation as depicted in Figure 3 consists of three logical building blocks: on lowest level, an adaptive CPU (ACPU) provides the actual reconfigurable computing hardware. The CPU divides into a standard RISC processor which can be enhanced by user-defined instructions provided by so-called functional elements (FE).

On top of the ACPU sits a VM responsible for mapping a uniform binary to the current system configuration. Upon processing an ASF exception, this VM decides whether this ASF should be mapped to a corresponding HW instruction (eventually forcing a reconfiguration cycle to load the dedicated ASI into hardware), or call an appropriate software implementation. This software representation can use either the static instruction set or, depending on the ACPU's configuration, less specific (but still application-supporting) instructions.

Since our example implementation focuses on power vs. performance optimization, we therefore introduce a specific logical instance called Performance-Power-Management (PPM). The PPM is physically distributed among program generation, where it aids to generate an initial ACPU configuration (and therefore the start configuration for the FEs), and, integrated into the VM, the run-time environment. During program execution, it evaluates the current system configuration by ASF attributes and additional parameters against given objective functions derived from user input and application attributes. The outcome of this evaluation process might lead to system reconfiguration.

4 Conclusion & Outlook

In this paper we briefly discussed two major problems arising from system reconfiguration on instruction level. These problems are binary compatibility and avoid violation of application requirements by the reconfiguration process itself. We delivered a light-weight and easy to implement solution for this dilemma, which is also suitable for use within embedded systems. Finally, we introduced an example architecture, currently under development, based on the described solutions.

References

1. P.M. Athanas and H.F. Silverman. Processor reconfiguration through instruction-set metamorphosis. In *IEEE Computer*, volume 26, pages 11–18. IEEE, Mar 1993.
2. J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th International IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21. IEEE, Apr 1997.
3. Xilinx Inc. Powerpc 405 processor block reference guide. `http://direct.xilinx.com/bvdocs/userguides/ug018.pdf`.
4. Xilinx Inc. Virtex-4 family overview. `http://direct.xilinx.com/bvdocs/publications/ds112.pdf`.
5. X.-P. Ling and H. Amano. WASMII: a data driven computer on a virtual hardware. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing*, pages 33–42. IEEE, Apr 1993.
6. M.J. Wirthlin and B.L. Hutchings. A dynamic instruction set computer. In *Proceedings of the 3rd International IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107. IEEE, Apr 1995.